Schema Polynomials and Applications [*]

Columbia University Computer Science Technical Report cucs-047-07

Kenneth A. Ross [†]
Columbia University
kar@cs.columbia.edu

Julia Stoyanovich [‡]
Columbia University
jds1 @cs.columbia.edu

ABSTRACT

Conceptual complexity is emerging as a new bottleneck as data-base developers, application developers, and database administrators struggle to design and comprehend large, complex schemas. The simplicity and conciseness of a schema depends critically on the idioms available to express the schema. We propose a formal conceptual schema representation language that combines different design formalisms, and allows schema manipulation that exposes the strengths of each of these formalisms. We demonstrate how the schema factorization framework can be used to generate relational, object-oriented, and faceted physical schemas, allowing a wider exploration of physical schema alternatives than traditional methodologies. We illustrate the potential practical benefits of schema factorization by showing that simple heuristics can significantly reduce the size of a real-world schema description. We also propose the use of schema polynomials to model and derive alternative representations for complex relationships with constraints.

1. INTRODUCTION

Physical data independence is the property that the physical schema used to store the data is independent of the logical schema used to conceptualize the same data. While textbooks advocate physical data independence, most database systems map conceptual structures such as relations directly to stored structures such as tables.

In a seminal paper [16], Tsatalos et al. enable higher degrees of physical data independence by allowing the stored structures to be defined via select-project-join views on the conceptual schema. Physical decisions are decoupled from the conceptual schema, and decisions like physical replication and nesting can be made without changing the conceptual schema. A wider range of physical representations can be considered in order to improve the database performance for a given workload.

Our viewpoint is similar to that of [16], in that we aim to decouple physical and conceptual data representations. Unlike [16], our focus is on optimizing the conceptual level representation to make the schema as simple as possible. Conceptual complexity is emerging as the new bottleneck as database developers, application developers (using SAP for example), and database administrators struggle to comprehend large, complex schemas.

There are many ways to measure schema simplicity, and we will not restrict ourselves to any one in particular. The general theme of simplicity, though, suggests that more concise representations of the same schema are better.

The conciseness of a schema depends critically on the idioms available to express the schema. As an analogy, consider the class of boolean expressions with conjunction and disjunction. It is well known that there are formulas that are compact in one representation (say disjunctive normal form) but exponentially bigger in another (say conjunctive normal form), and vice versa.

We argue that a similar idiomatic conflict occurs for schema design. Three well-known formalisms for writing schemas are the relational, the object-oriented, and the faceted approaches. Each of these is very good at expressing certain classes of schema, but poor at expressing certain other classes of schema. Object-oriented schemas allow the factorization of common attributes into an inheritance hierarchy. Faceted schemas allow the orthogonal composition of many independent attributes; in an e-commerce application for example, a product may be independently classified by size, by brand, by color, etc.

Traditionally, one has been limited to just one of these design methodologies when creating schemas. Even if part of the schema is compactly represented using that methodology, other parts may not be. Our aim is to define a higher-level conceptual data representation language that allows the schema to be manipulated in ways that expose the strengths of each of these data modeling approaches. If a schema has parts that are compact when expressed according to one approach, then that part of the schema should be expressed using that approach, even if other parts of the schema use a different approach.

[*] A shorter version of this report will appear at the International Conference on Extending Database Technology(EDBT), March 2008.

[†] Supported by National Science Foundation grants IIS-0121239, IIS-0534389 and National Institute of Health grant 5 U54 CA121852-03.

[‡] Supported by National Science Foundation grant IIS-0121239 and National Institute of Health grant 5 U54 CA121852-03.

Consider for example the database schema of an online bookstore. This schema may be used to store the inventory (i.e. different kinds of books and periodicals), allow for faceted browsing of the inventory by several independent dimensions, e.g. by author, topic, and price, and record information about customers and orders. Designing such a schema currently requires that the architect choose a formalism, e.g. relational, faceted, or XML, design the entire schema using that formalism, and then translate the schema into its physical representation, either manually or with the help of design tools. In our example, it may be most natural to model the inventory sub-schema using the object-oriented paradigm, organize some aspects of the inventory in an orthogonal hierarchy using the faceted model, but retain purely relational representation of customers and orders. The choice of a different formalism for each part of the schema will result in a schema that is easier to understand and is more concise than if a single formalism were chosen to represent the schema in its entirety.

The benefits of conciseness are twofold. First, as argued above, concise schemas tend to be simpler, and easier for users to understand. Second, since physical design usually starts with a conceptual level description, simpler physical designs can be produced. Sometimes these simpler physical designs have better performance than alternative designs.

We propose a formal language for representing schemas, and derive a set of properties that allow one to manipulate schema expressions while preserving the set of representable tuples. We propose that standard design methodologies such as entity-relationship modeling [15], modeling in UML [13, 8], faceted modeling [20] and relational normalization [15], produce outputs in this conceptual framework rather than going directly to a physical relational schema. By doing so, one obtains the following advantages:

- One can explore a variety of physical representations obtained using different equivalent expressions for a schema.
- One can derive a formally unambiguous logical representation that allows transformations that are not straightforward for sets of physical tables, such as attribute factorization and subtraction.
- Different users can orient the schema to their own points of view. One user may impose an inheritance hierarchy (e.g., factoring out the location of manufacture) while another may impose an alternative hierarchy (e.g., factoring out the product-type). These conceptual views are compatible, and can be automatically translated into the chosen physical representation.
- Users can project out parts of the schema that they are not interested in. The remaining portion can be simplified using various equivalences, leading to a description that is much easier to understand than a schema with hundreds of tables and thousands of columns.

Example 1.1. *Consider the following schema expression.*

$$ABCD + ABG + ACD + AG + BCH \tag{1}$$

When we write $ABCD$ in Expression 1 we mean that tuples with attribute names A, B, C, and D are valid for this schema. When we write $S_1 + S_2$ we mean that a tuple is valid for either S_1 or S_2. As written, the schema could be mapped to a collection of five stored tables **ABCD**, **ABG**, **ACD**, **AG**, *and* **BCH**. *We use bold script to describe physical level structures such as tables, and math script for conceptual level expressions. We could alternatively factorize the schema into the representation*

$$A(B+1)(CD+G) + BCH \tag{2}$$

which has fewer syntactic elements than Expression 1. Common occurrences of some attributes have been factored out. The $B+1$ subexpression allows tuples having either a valid or a null value for attribute B.

Expression 2 does not have a direct relational interpretation, since there is no relational construct to express $CD+G$. Nevertheless, one can map this expression to a set of tables for storage as follows: Add a new unique identifier column (say I) to represent the link between the $A(B+1)$ subexpression and the $(CD+G)$ subexpression, to yield tables **ABI**, **ICD**, **IG** *and* **BCH**, *where B can be null in* **ABI**. *This transformation is correct only with constraints on the new attribute I. The constraint for I in* **ABI** *says that the value of I must appear in exactly one of* **ICD** *and* **IG**. *There must also be foreign key constraints on I into* **ABI** *from both* **ICD** *and* **IG**.

An alternative rewriting of Expression 2 is

$$A(B+1)[(CD+1)(G+1) - CDG - 1] + BCH \tag{3}$$

The subtractions in Expression 3 are interpreted as constraints stating that (a) C, D, and G cannot all be non-null, and (b) C, D, and G cannot all be null. This expression can be mapped to a physical schema **ABCDG**, **BCH**, *where* **ABCDG** *has various null/not-null constraints on B, C, D, and G.*

If one cared only about attributes A, B, and C in this schema, one could project out the other attributes to obtain from 1 the simpler expression $ABC + AB + AC + A + BC$, which could again be factorized in various ways. □

Example 1.2. *Yet another factorization of Expression 1 is*

$$C(ABD + AD + BH) + ABG + AG \tag{4}$$

This would be a natural factorization in a hierarchical or object-oriented view in which C is the primary dimension of classification and is inherited from a higher-level entity. Unlike traditional object-oriented data modeling that imposes a single hierarchical structure, our approach allows different users with different points of view to "orient" the schema according to the dimensions they are most interested in. In such a case, the measure of simplicity for a user might give added weight to the absence of redundancy for the C attribute in Expression 4, since the user cares more about C than the other attributes. □

Example 1.3. *As a final example, consider the expression*

$$ABC + AB + BC + AC + A + B + C \tag{5}$$

We might rewrite this expression as

$$(A+1)(B+1)(C+1) - 1. \tag{6}$$

This expression has the quality of a faceted classification [20]. A tuple can independently have an A, B, and/or C attribute, as long as it has at least one of them. One physical *representation of this schema would be a single table* **ABC** *with constraints allowing any combination of null/not-null values except all three being null. Another physical representation, using a transformation similar to one used in Example 1.1, would use three tables* **AI**, **BI**, *and* **CI** *with no attributes allowed to be null. This second physical representation is "faceted" in the sense that it allows the storage of A, B and C to happen independently.* □

Our choice of polynomial-like notation, with addition, multiplication, and subtraction is deliberate. As we shall demonstrate formally, addition and multiplication satisfy the familiar commutativity and distributivity laws (although subtraction does not satisfy all such laws), and all transformations described above are valid for schema manipulation.

Our schema manipulation allows the use of nulls[1] as a legitimate design construct when it simplifies the schema, as in Examples 1.1 and 1.3. Current design methodologies sometimes frown on the use of nulls, stating that they should be avoided or used only as a last resort. A dogmatic prohibition of nulls can be counterproductive, and a judicious use of nulls can lead to better conceptual and physical schemas. Our methods allow for a *disciplined* use of nulls in order to improve some quantitative measure of schema simplicity.

Data redundancy is a well-studied problem in the domain of relational database systems. A number of normal forms have been developed to address data redundancy and various kinds of anomalies. Normalization can be incorporated into our framework; see Section 2.2. Our work is complementary to traditional normalization in that we try to minimize the size of the conceptual-level schema representation, rather than minimizing data redundancy.

In schema factorization we generate a variety of possible equivalent expressions that make the schema concise or orient the schema to the point of view of the user. Another application of schema polynomials is the representation of complex relationships in the presence of constraints. Given entities and their inheritance hierarchy as input, *relationship factorization* generates a variety of possible equivalent expressions that take constraints into account, and orient the relationships to the user's point of view.

The rest of this paper is organized as follows. We formally define the proposed schema description language in Section 2, and describe a class of rewrite rules that allow automatic generation of relational, object-oriented, and faceted physical representations in Section 3. The complexity of finding minimal schema representations is discussed in Section 4, followed by experimental results demonstrating the potential of schema factorization in Section 5. We develop an application of schema polynomials to relationships and constraints in Section 6. Related work is outlined in Section 7. We discuss future work and conclude in Section 8.

2. SCHEMA EXPRESSIONS

The basic notion in describing schemas is the *attribute*. We shall assume that two instances of the same attribute name denote the same concept. Thus we avoid generic attribute labels like "name", and prefer specific labels like "person-name" or "company-name". We assume without further comment that attributes have data types from some type language, and that data values in the database have the correct types with respect to the attribute's data types. We will use the symbols A, B, C, etc. to denote attributes.

2.1 Tuple Descriptors and Tuples

Definition 2.1. *A* tuple-descriptor T *is defined recursively using the following context free grammar:*

$$T \quad ::= \quad 0 \mid 1 \mid A \mid B \mid C \mid \ \dots \ \mid TT \mid T+T \mid T-T$$

We will describe the concatenation TT as multiplication. *A, B, C, ... are the attribute names, 1 is a special symbol denoting an empty product (i.e., the empty set of attributes), and 0 is a special symbol denoting an empty sum.* □

A tuple-descriptor is a schema-level constraint that describes the set of valid tuples in the database. This idea is formalized below.

Definition 2.2. *A* tuple *is a set of (attribute, value) pairs (a_i, v_i), with no attribute name a_i appearing more than once. The empty tuple ϵ (with no attributes) is permitted. A tuple t matches a tuple-descriptor T if and only if:*

1. *$T = 1$ and $t = \epsilon$.*
2. *T is a single attribute, and t contains a single pair (T, v) for some value v.*
3. *$T = T_1T_2$ and there exist t_1 and t_2 such that $t_1 \cup t_2 = t$, t_1 matches T_1, and t_2 matches T_2.*
4. *$T = T_1 + T_2$, and t matches either T_1 or T_2.*
5. *$T = T_1 - T_2$, t matches T_1, and t does not match T_2.*

Note that no tuples match the tuple-descriptor 0. □

Definition 2.3. *If S is a set of tuples, we say S* matches *a tuple descriptor T if every tuple in S matches T. We denote by $\mathcal{S}(T)$ the set of all tuples that match T.* □

Definition 2.1 provides the syntax for tuple descriptors. The semantics of a tuple-descriptor T is the collection of tuples $\mathcal{S}(T)$ that match T according to Definition 2.2. We write $T_1 = T_2$ if T_1 and T_2 admit exactly the same tuples, even if T_1 and T_2 are not syntactically identical. Based on Definition 2.2, we identify the following properties of tuple-descriptors.

Lemma 2.1. *Given tuple-descriptors T_1 and T_2*

- *$T_1 - T_2 = 0$ if and only if $\mathcal{S}(T_1) \subseteq \mathcal{S}(T_2)$.*
- *$T_1 - T_2 = T_1$ if and only if $\mathcal{S}(T_1) \cap \mathcal{S}(T_2) = \emptyset$.*

Proof. *Both properties follow directly from set algebra.*

- *$\mathcal{S}(T_1 - T_2) = \mathcal{S}(T_1) - \mathcal{S}(T_2) = \emptyset$ if and only if $\mathcal{S}(T_1) \subseteq \mathcal{S}(T_2)$.*
- *$\mathcal{S}(T_1 - T_2) = \mathcal{S}(T_1) - \mathcal{S}(T_2) = \mathcal{S}(T_1)$ if and only if $\mathcal{S}(T_1) \cap \mathcal{S}(T_2) = \emptyset$.*

□

[1] Nulls have been used with a variety of interpretations. For us, null means "not applicable". Very different designs would apply if one was to interpret null as "value at present unknown" [14] or "no information" [5].

LEMMA 2.2. *Addition and multiplication on tuple-descriptors are commutative and associative. Addition and subtraction are idempotent, i.e., $(T_1 - T_2) - T_2 = T_1 - T_2$. Multiplication of attributes is idempotent.* 1 *is a multiplicative identity,* 0 *is an additive identity and a right-identity for subtraction, $0T = T0 = 0$ for all T, and multiplication distributes over addition.*
Proof.

- *Commutativity of addition follows directly from Definition 2.2: $T = T_1 + T_2 = T_2 + T_1$.*

- *Commutativity of multiplication follows directly from Definition 2.2: $T = T_1T_2 = T_2T_1$.*

- *Associativity of addition follows from associativity of set-union:*
$\mathcal{S}((T_1 + T_2) + T_3) = \mathcal{S}(T_1 + T_2) \cup \mathcal{S}(T_3) = (\mathcal{S}(T_1) \cup \mathcal{S}(T_2)) \cup \mathcal{S}(T_3) = \mathcal{S}(T_1) \cup (\mathcal{S}(T_2) \cup \mathcal{S}(T_3)) = \mathcal{S}(T_1 + (T_2 + T_3))$

- *We show associativity of multiplication as follows. Consider a tuple t that matches a tuple-descriptor $T = (T_1T_2)T_3$. Since t matches T, then there exist tuples t_{12} and t_3 such that $t_{12} \cup t_3 = t$, t_{12} matches T_1T_2 and t_3 matches T_3. Since t_{12} matches T_1T_2, then there exist tuples t_1 and t_2 such that t_1 matches T_1 and t_2 matches T_2 and $t_{12} = t_1 \cup t_2$.*
But then there also exists a tuple $t_{23} = t_2 \cup t_3$ that matches T_2T_3. Note that t_{23} does not have two different values for the same attribute, since $t_{23} \subseteq t$. Since t_1 matches T_1 and t_{23} matches T_2T_3, then $t = t_1 \cup t_2 \cup t_3$ matches $T_1(T_2T_3)$.

- *We show idempotence of multiplication over attributes as follows. $TT \subseteq T$ by Definition 2.2 part 3, choosing $t_1 = t_2 = t$. For a single attribute T, if t_1 matches T and t_2 matches T then t_1 and t_2 must share the same attribute name. If they have different values, then $t_1 \cup t_2$ is not a tuple. If they have the same value, then $t_1 = t_2$ matches T.*

- *Idempotence of addition follows directly from idempotence of set-union:*
$\mathcal{S}(T + T) = \mathcal{S}(T) \cup \mathcal{S}(T) = \mathcal{S}(T)$.

- *Idempotence of subtraction follows from itempotence of set-difference:*
$\mathcal{S}(T_1 - T_2 - T_2) = \mathcal{S}(T_1) - \mathcal{S}(T_2) - \mathcal{S}(T_2) = \mathcal{S}(T_1) - \mathcal{S}(T_2) = \mathcal{S}(T_1 - T_2)$

- *We show distributivity of multiplication over addition as follows.*
We first demonstrate that $T_1(T_2 + T_3) \subseteq T_1T_2 + T_1T_3$. Consider a tuple t that matches $T_1(T_2 + T_3)$. There exist tuples t_1 and t_{23} such that t_1 matches T_1 and t_{23} matches T_2 or T_3, and $t = t_1 \cup t_{23}$. We choose T_2 or T_3 depending on which of the two tuple-descriptors t_{23} matches; say, without loss of generality, that t_{23} matches T_2. Then $t_1 \cup t_{23}$ matches $T_1T_2 \subseteq T_1T_2 + T_1T_3$.
Let us now show that $T_1T_2 + T_1T_3 \subseteq T_1(T_2 + T_3)$. If tuple t matches T_1T_2, then there exist t_1 and t_2 such that t_1 matches T_1 and t_2 matches T_2. Since t_2 matches T_2, it also matches $T_2 + T_3$. So t matches $T_1(T_2 + T_3)$. A similar analysis holds when t matches T_1T_3.

- *The fact that* 1 *is a multiplicative identity follows directly from the definition: T and $T1$ admit exactly the same set of tuples, since $t \cup \epsilon = t$.*

- *The fact that* 0 *is an additive identity also follows from the definition.*
$\mathcal{S}(T + 0) = \mathcal{S}(T) \cup \mathcal{S}(0) = \mathcal{S}(T) + \emptyset = \mathcal{S}(T)$.

- *Finally, we see that* 0 *is the right identity of subtraction because*
$\mathcal{S}(T - 0) = \mathcal{S}(T) - \mathcal{S}(0) = \mathcal{S}(T) - \emptyset = \mathcal{S}(T)$.

□

The idempotence of addition means that it does not matter how many times a particular combination of attributes appears in a sum; what matters is just whether the combination appears at all.[2] The idempotence of multiplication means that repeated instances of an attribute in a product term of a tuple-descriptor are not significant. In other words, saying twice that a tuple has an attribute is the same as saying it once. Note that multiplication of tuple-descriptors is not idempotent: $(A + BC)(A + BC) = A + BC + ABC \neq A + BC$. Subtraction is not associative with addition, since $A + (A - A) = A \neq (A + A) - A = 0$. However, we can get a limited form of associativity according to the following lemma.

LEMMA 2.3. *Let T_1, T_2, and T_3 be tuple-descriptors. $T_1 + (T_2 - T_3) = (T_1 + T_2) - T_3$ if and only if $T_3 - T_1 = T_3$.*
Proof. *By Definition 2.2, $\mathcal{S}(T_3) - \mathcal{S}(T_1) = \mathcal{S}(T_3)$. Using this, and the property of set algebra that $(\mathcal{B} - \mathcal{A}) \cup \mathcal{C} = (\mathcal{B} \cup \mathcal{C}) - (\mathcal{A} - \mathcal{C})$, we have:*

$$
\begin{aligned}
\mathcal{S}(T_1 + (T_2 - T_3)) &= \mathcal{S}(T_1) \cup (\mathcal{S}(T_2) - \mathcal{S}(T_3)) \\
&= (\mathcal{S}(T_1) \cup \mathcal{S}(T_2)) - (\mathcal{S}(T_3) - \mathcal{S}(T_1)) \\
&= (\mathcal{S}(T_1) \cup \mathcal{S}(T_2)) - \mathcal{S}(T_3) \\
&= \mathcal{S}((T_1 + T_2) - T_3)
\end{aligned}
$$

□

Multiplication does not distribute over subtraction, since, for example, $B(AB - A) = AB \neq (AB - AB) = 0$.

DEFINITION 2.4. *A tuple-descriptor is in* additive normal form *if it has no subtractions, and consists of a sum of products of attributes: In each product, no attribute is mentioned more than once, and there are no repeated product terms.* □

LEMMA 2.4. *Every tuple-descriptor has an equivalent tuple-descriptor in additive normal form. The additive normal form representation is unique up to the order of terms in sums and products.*
Proof of existence*: For expressions without subtractions, the transformation is standard, and is similar to a disjunctive normal-form transformation using De Morgan's Laws. Subtractions can be eliminated inductively from inner subexpressions outwards. Given $T_1 - T_2$ where neither T_1 nor T_2 contain subtractions, construct additive normal form expressions T_1' and T_2' equivalent to T_1 and T_2 respectively. Eliminate from T_1' any product terms appearing in T_2'. The result*

[2] There is no notion of a "relation name" for tuples. Any semantic information that might be embedded in a table name in a relational database (such as calling a table "part-time-employees") has to be provided instead via attribute names and values.

is an additive normal form representation of $T_1 - T_2$.
Proof of uniqueness*: Suppose that, in addition to* T'*, there existed another tuple-descriptor* T''*, such that* T *is equivalent to* T''*,* T'' *is in additive normal form, and* T' *and* T'' *are not syntactically identical. The transformation to additive-normal form is correct, in the sense that* T' *and* T'' *are both equivalent to* T*:* $\mathcal{S}(T) = \mathcal{S}(T') = \mathcal{S}(T'')$*. By definition of additive normal form, a tuple* t *matches a tuple-descriptor in additive-normal form if it matches exactly one additive term in the descriptor. If* T' *and* T'' *are not syntactically identical, then let us suppose, without loss of generality, that there exists an additive term* T_1 *in* T' *but not in* T''*. However,* $\mathcal{S}(T') = \mathcal{S}(T'') + \mathcal{S}(T_1) = \mathcal{S}(T'')$ *if and only if* $\mathcal{S}(T_1) = \emptyset$*. Then, by Definition 2.2,* $T_1 = \epsilon$*.* □

Definition 2.5. *Given tuple-descriptors* T_1 *and* T_2*, we say that* T_1 subsumes T_2 *if, for every tuple* t *such that* t *matches* T_2*,* t *also matches* T_1*.* □

Lemma 2.5. *A tuple-descriptor* T_1 subsumes *a tuple-descriptor* T_2 *if and only if, in the additive normal forms* T_1' *and* T_2' *of* T_1 *and* T_2 *respectively, every product term appearing in* T_2' *also appears in* T_1'*.*
Proof. *Given that* $\mathcal{S}(T_2) \subseteq \mathcal{S}(T_1)$*, then also* $\mathcal{S}(T_2') \subseteq \mathcal{S}(T_1')$*. Tuple* t *matches* T_2' *if it matches exactly one term* S *in* T_2'*. Since* t *also matches* T_1'*, the term* S *must be present in* T_1'*.*
□

Definition 2.6. *Given tuple-descriptors* T_1 *and* T_2*, we define* $T_1 \cap T_2$ *to mean the set of tuples* t *that match both* T_1 *and* T_2*.* □

Lemma 2.6. *For any pair of tuple-descriptors* T_1 *and* T_2*, there exists a tuple-descriptor* T *equivalent to* $T_1 \cap T_2$*.*
Proof. T *can be constructed by taking the additive normal forms of* T_1 *and* T_2 *and retaining only product terms that appear in both. If there are no common product terms,* $T_1 \cap T_2 = 0$*. Consider expressions* T_1' *and* T_2' *that represent additive normal forms of* T_1 *and* T_2*, respectively. A tuple* t *matches* T_1 *iff it matches exactly one additive term in* T_1'*, call it* S_1*. Likewise,* t *matches a single additive term in* T_2'*, call it* S_2*. Since* t *matches both* S_1 *and* S_2*, and since* S_1 *and* S_2 *are in additive normal form, then* S_1 *must be syntactically identical to* S_2*.* □

As a result of Lemmas 2.5 and 2.6, it is possible to syntactically determine the subsumption and intersection relationships between tuple-descriptors.

When a user is interested in only a subset of attributes, the schema could be projected onto just that subset. For expressions without subtraction, there is a simple transformation: replace all instances of unwanted attributes by 1. This transformation is not valid for expressions with subtraction, so an equivalent subtraction-free tuple-descriptor (such as its additive normal form) should be derived before projecting out attributes.

We remark that we were tempted to include division in our formalism to represent overriding. One might write $A(B + C + A^{-1}D)$ to represent the overriding of attribute A by D in a subclass of the class defining attribute A. However, adding division would make multiplication non-associative, since $(AA)A^{-1} = 1 \neq A(AA^{-1}) = A$. In future work, we plan to examine syntactic restrictions on tuple-descriptors that might avoid such problems.

2.2 Relationship to Other Schema Design Formalisms

Our formalism bases its semantics on the set of tuples that a tuple-descriptor permits. But where does the “correct” set of tuples for an application come from? This is an important question, because a poor choice of tuple types can lead to substantial data redundancy and insertion/deletion/update anomalies.

We propose to take as input to our conceptual framework, the output of a conventional design tool. Instead of generating relations, textbook ER-diagram-to-table transformations can be recast as transformations from ER-diagrams to tuple-descriptors. (Some ER-diagram elements such as cardinality constraints are not representable as tuple-descriptors, and would need to be re-examined once a physical design is chosen.) Similarly, instead of generating tables directly, a relational normalization algorithm could generate a tuple-descriptor. For example, if the Boyce-Codd normal form (BCNF) of a schema was $\{ABC, CDE, EF\}$, then we would take the expression $ABC + CDE + EF$ as the corresponding tuple-descriptor.

Two different normalizations of the same initial set of attributes may lead to nonequivalent tuple-descriptors. Consider the simple schema ABC with functional dependencies $A \rightarrow B$ and $A \rightarrow C$. $\{ABC\}$ is in BCNF, as is the schema $\{AB, AC\}$. But ABC and $AB + AC$ are not equivalent tuple-descriptors. $\{ABC\}$ is the superior normalization because $\{AB, AC\}$ needs additional foreign key constraints to embody the same information. If B and C are permitted to be null, but A was not, then we would map $\{ABC\}$ to $A(B + 1)(C + 1)$ instead.

3. PHYSICAL REPRESENTATIONS

Given a schema expression, we describe a class of rewrite rules that allows the automatic generation of physical representations. Multiple rewrite rules may be valid for a single expression, leading to multiple options for the physical representation. We assume that a physical column must be non-null, unless nulls are explicitly allowed via a constraint. In some of the rewrite rules, a new “identifier attribute” is added to the schema. A constraint stating that this attribute is unique should be attached to every physical structure in which it appears. We distinguish between subexpressions T that are part of the original expression, and new identifier attributes I that are introduced during the rewrite process.

While we have not precisely defined the class of constraints that are allowed, all of the constraints mentioned in the rewrite rules are easily expressed in conventional relational constraint formalisms. Some of the rewrite rules place constraints on the representations of subexpressions. Since these subexpressions may themselves be further decomposed, such constraints may need to be expressed as constraints on views over the decomposed representations.

3.1 Relational Physical Representation

For the relational rewrite rules, we assume that the physical representation language provides the table as the abstract storage type. There are several data structures that could be used to represent a table, but the choice of data structure is orthogonal to the choice of which tables to build.

Rewrite Rule 3.1. *Given an expression equal to* 0*, generate the empty physical representation.* □

Rewrite Rule 3.1 handles cases where a recursive application of other rules requires a representation for an expression like $1 - 1$.

Rewrite Rule 3.2.

(a) *Given an expression of the form $A_1 \dots A_n(B_1 + 1) \dots (B_m + 1)$, construct a table with $n+m$ columns matching the attributes $A_1, \dots, A_n, B_1, \dots, B_m$, and a constraint that allows each of $B_1, \dots, B_m$ to be null.*

(b) *Given an expression of the form $IA_1 \dots A_n(B_1 + 1) \dots (B_m + 1)$, construct a table with $n + m + 1$ columns matching the attributes $I, A_1, \dots, A_n, B_1, \dots, B_m$, and a constraint that allows each of $B_1, \dots, B_m$ to be null.* □

Rewrite Rule 3.3.

(a) *Given an expression $T_1 + T_2$, construct the union of the physical representations R_1 and R_2 of T_1 and T_2 respectively.*

(b) *Given an expression $I(T_1+T_2)$, where I is an identifier attribute, construct the union of the physical representations R_1 and R_2 of IT_1 and IT_2 respectively, and add a constraint requiring that the values of I in R_1 and R_2 are disjoint.* □

Note that in Rewrite Rule 3.3, when $T_1 \cap T_2$ is non-empty, we have some flexibility about whether a tuple matching $T_1 \cap T_2$ is stored in R_1, R_2, or both.

Example 3.1. *Consider the expression*

$$ABCDE + ABCE + ABDE + ABE + AE + BE + E$$

which we might choose to factorize as $E(A + 1)(B + 1) + EAB(C + 1)(D + 1)$. Using Rewrite Rules 3.3 and 3.2, we obtain the schema **EAB**, **EABCD** *with constraints: (a) Each of A and B may be null in* **EAB**; *(b) Each of C and D may be null in* **EABCD**; *(c) Every non-null EAB triple in* **EAB** *must appear in* **EABCD** *and vice versa.* □

Rewrite Rule 3.4.

(a) *Given an expression of the form $T_1 - T_2$, let T_2' be the additive normal form of T_2. Construct the physical representation of T_1, and add a constraint that disallows each of the combinations of null/non-null values described by the product terms of T_2'.*

(b) *Given an expression of the form $I(T_1 - T_2)$, let T_2' be the additive normal form of IT_2. Construct the physical representation of IT_1, and add a constraint that disallows each of the combinations of null/non-null values described by the product terms of T_2'.* □

Example 3.2. *Consider the factorized expression $A(B+1)(C+1) - AC$. Using Rewrite Rules 3.4 and 3.2, we would generate a physical schema* **ABC** *in which B and C are allowed to be null, and if B is null, C must be null.* □

Rewrite Rule 3.5. *Given an expression T without any identifier attributes, let I be a new identifier attribute, and recursively construct the physical representation of IT.* □

Rewrite Rule 3.6. *Given an expression of the form IT_1T_2, where the sets of attributes that appear in T_1 and T_2 are disjoint, recursively construct physical representations R_1 and R_2 for IT_1 and IT_2 respectively. Add an integrity constraint stating that any tuple in R_1 must have exactly one tuple in R_2 matching on the I column, and vice-versa.* □

Rewrite Rule 3.7. *Given an expression of the form $I(T_1+1)T_2$, where the sets of attributes that appear in T_1 and T_2 are disjoint, recursively construct physical representations R_1 and R_2 for IT_1 and IT_2 respectively. Add an integrity constraint stating that any tuple in R_1 must have exactly one tuple in R_2 matching on the I column.* □

Rewrite Rule 3.8. *Given an expression of the form $I((T_1 + 1)T_2 - 1)$, where the sets of attributes that appear in T_1 and T_2 are disjoint, and where T_2 subsumes 1, recursively construct physical representations for IT_1 and $I(T_2 - 1)$ respectively. (No constraints are added.)* □

Example 3.3. *Consider again the expression of Example 3.1 which we might choose to factorize instead as $E(A+1)(B+1)+EAB[(C+1)(D+1)-1]$. Using Rewrite Rules 3.3, 3.2, 3.5, 3.6, 3.8, and 3.1, we would obtain the schema* **EAB**, **EABI**, **IC**, **ID** *with the following constraints: (a) Each of A and B may be null in* **EAB**; *(b) Every I value in* **EABI** *must appear in at least one of* **IC** *and* **ID**; *(c) Every I value in* **IC** *must appear in* **EABI**; *(d) Every I value in* **ID** *must appear in* **EABI**; *(e) Column I is unique in each table containing I.* □

Comparing Examples 3.1 and 3.3, we see two quite different physical representations for the same initial expression. While the factorization of Example 3.1 was more concise, and the physical representation required fewer tables, the generated physical representation contains some data redundancy: EAB triples are stored twice. This redundancy could improve performance (some queries are faster because one table can be consulted rather than two) or worsen performance (updates need to modify two copies), and the net benefit depends on the workload. Note that we are not concerned with other typical pitfalls of redundancy, such as inconsistent updates. Insertions/deletions would be performed on the conceptual schema and automatically translated into modifications to the data in the physical schema. As long as the transformations are correct, inconsistent copies cannot arise.

Lemma 3.1. *The collection of rewrite rules above is correct, in the sense that a tuple set S matches a tuple-descriptor if and only if S, extended with unique values of identifier attributes, can be faithfully represented in the generated set of tables with constraints.*

Proof. *The proof is by induction on the size of the expression, with a separate argument for each rewrite rule:*

- *Rule 3.1 trivial.*
- *Rule 3.2 Any tuple matching $A_1 \dots A_n(B_1+1) \dots (B_m+1)$ must include all attributes A_i, may optionally include any number of the B_i attributes, and cannot include any other attributes. This is exactly the class of tuples representable in a table $A_1 \dots A_nB_1 \dots B_m$ with the B_i columns allowed to be null.*

- *Rule 3.3(a): Let S_1 be the fragment of S satisfying T_1, and similarly for S_2 and T_2. By the induction hypothesis, the physical representations R_1 and R_2 can faithfully represent S_1 and S_2 respectively, and so this pair of tables together do indeed represent tuples that match either T_1 or T_2.*

- *Rule 3.3(b): Similar to Rule 3.3(a). The disjointness condition prevents the combination of the two physical tables from violating the property that I is an identifier attribute.*

- *Rule 3.4: straightforward.*

- *Rule 3.5: trivial.*

- *Rule 3.6: Because T_1 and T_2 have disjoint attributes, we can partition each tuple $t \in S$ into parts t_1 and t_2 such that t_i mentions only the attributes in IT_i, and $t = t_1 \cup t_2$. Let S_1 be the set of tuple parts t_1, and similarly for S_2. By the induction hypothesis, R_1 faithfully represents S_1 and R_2 faithfully represents S_2. S can be reconstructed as the join of R_1 and R_2 on I, which is lossless because I is a key attribute. The constraints between R_1 and R_2 prevent the possibility of a dangling tuple, e.g., an S_1 tuple without a matching S_2 tuple.*

- *Rule 3.7: Similar to Rule 3.6. The difference here is that it is possible to have an S_2 tuple without a matching S_1 tuple.*

- *Rule 3.8: Similar to Rule 3.6.*

□

LEMMA 3.2. *The collection of rewrite rules above, in conjunction with the transformation to additive normal form, is complete, in the sense that every tuple-descriptor can be rewritten into a corresponding relational representation.*

Proof. *Tuple-descriptors are generated by a context free grammar in Definition 2.1. Let us show how each production of the grammar is handled by the rewrite rules. We apply rewrite rules recursively, using an inductive argument on the number of operators in the expression. Given a tuple-descriptor T, rewrite rules are considered in the following order:*

- *If $T = 0$ apply Rewrite Rule 3.1.*

- *If $T = 1$ or T is of the form $A_1 \ldots A_n(B_1{+}1) \ldots (B_m{+}1)$ apply Rewrite Rule 3.2.*

- *If T is in additive normal form, apply Rewrite Rule 3.3.*

- *If T is of the form $T_1 - T_2$ apply Rewrite Rule 3.4.*

- *If T does not contain any identifier attributes, apply Rewrite Rule 3.5, transforming the tuple-descriptor to IT.*

- *If T is of the form $I((T_1 + 1)T_2 - 1)$ apply Rewrite Rule 3.8.*

- *Consider now a tuple-descriptor $T = IT_1T_2$ such that the set of attributes that appear in T_1 and T_2 are disjoint. If T is of the form $I(T_1 + 1)T_2$ apply Rewrite Rule 3.7. Otherwise apply Rewrite Rule 3.6.*

- *If T is of the form T_1T_2 and the sets of attributes in T_1 and T_2 are not disjoint, we must transform T into additive normal form, and then apply Rewrite Rules 3.3 and 3.2 as described above.*

□

3.2 Object-Oriented Physical Representation

In an object-oriented system, the basic abstraction is the *class*. For each attribute A in the system, a special class has_A is defined with a single attribute A.[3] A class C_1 can inherit from another class C_2, in which case every attribute defined for C_2 is also valid for C_1. The inheritance relation between classes must be acyclic, and the special has_A classes do not inherit from other classes. A class C admits a tuple t as an instance if and only if t and C have exactly the same set of attributes. A class can be declared as *virtual* meaning that it is not permitted to have instances.

DEFINITION 3.1. *Let T be a tuple-descriptor. We say T is* simple *if it can be generated by the following context-free grammar.*

$$\begin{array}{lcl} S & ::= & A \mid B \mid C \mid \ldots \\ T & ::= & 1 \mid S \mid ST \mid T + T \end{array}$$

□

A simple tuple-descriptor is one that has no subtraction, and in which every left-multiplier is a single attribute. Since the additive normal form is simple, every tuple-descriptor has at least one equivalent simple representation.

A simple tuple-descriptor can be rewritten into an object-oriented physical representation using the following rewrite rules. During rewriting, we keep track of a "parent set" P that links a subclass to its containing superclass(es). The initial parent set of the complete expression is empty. We assume that all classes are virtual unless explicitly marked as "nonvirtual" by one of the rewrite rules.

REWRITE RULE 3.9. *Given the expression 1, and a parent set P, mark each class in the parent set as nonvirtual.* □

REWRITE RULE 3.10. *Given a tuple-descriptor T of the form $A_1, \ldots, A_n$, and a parent set P, create a new class C. C inherits from each class in P. C also inherits from each of the classes has_$A_1, \ldots,$ has_A_n. Mark C as non-virtual.* □

REWRITE RULE 3.11. *Given a tuple-descriptor T of the form $A_1, \ldots, A_nT_1$, and a parent set P, create a new class C. C inherits from each class in P. C also inherits from each of the classes has_$A_1, \ldots,$ has_A_n. The physical representation of T_1 is recursively constructed with parent set $\{C\}$.* □

REWRITE RULE 3.12. *Given an expression T of the form $T_1 + T_2$, and a parent set P, recursively construct physical representations of each T_1 and T_2, both with parent set P.* □

[3] The point of these has_A classes is to ensure that attributes with the same name inherited on different paths are the same attribute.

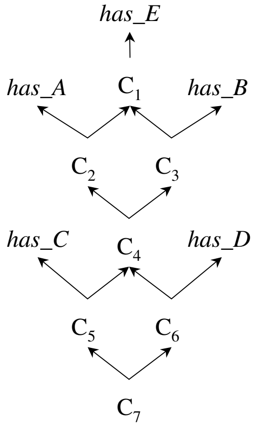

Figure 1: Object-oriented schema with multiple inheritance

Rewrite Rule 3.13. *Given an expression T of the form $S_1T_1+S_2T_2+S_1S_2T_3$, where S_1 and S_2 are products of single attributes, and a parent set P, proceed as follows. Construct classes C_1 and C_2, both of which inherit from all classes in P. Construct a class C_3 that inherits from both C_1 and C_2. For each i, if attribute A_i appears in S_1, then C_1 inherits from has_A_i, and if attribute A_i appears in S_2, then C_2 inherits from has_A_i. Recursively construct the physical representation of T_1 with parent set $\{C_1\}$. Recursively construct the physical representation of T_2 with parent set $\{C_2\}$. Recursively construct the physical representation of T_3 with parent set $\{C_3\}$.* □

Rewrite Rule 3.13 identifies one kind of multiple inheritance. There are many other patterns that could be detected as multiple inheritance, and corresponding rewrite rules could be derived if such patterns were common.

Example 3.4. *Consider once more the expression of Example 3.1 which we now factorize as the simple expression*

$$E[1+A+B+AB(1+C+D+CD)]$$

Using Rewrite Rules 3.11, 3.9, 3.10 and 3.13, we would obtain the double-diamond schema shown in Figure 1, with all classes C_i nonvirtual. □

Lemma 3.3. *The collection of rewrite rules above is correct, in the sense that a tuple set $\mathcal{S}$ matches a tuple-descriptor if and only if $\mathcal{S}$ can be faithfully represented by the set of nonvirtual classes in the generated physical schema.*

Proof. *First observe that the parent set for any rewrite construction is either empty (for the initial rewrite application), or contains a single class (for recursively spawned rewrite applications). At any intermediate point in the construction, a class C will inherit from various other classes. Once the inheritance relationship for C is fixed by a rewrite rule, subsequent rewrite rules do not change the inheritance structure of C, since they work either on subclasses of C, or on classes on separate branches of the hierarchy. Thus it makes sense to define att(C), the set of attributes inherited by C, at any point of the construction after C is created.*

Consider an expression T that has not yet been rewritten at a certain point during the computation, and let $\{C\}$ be the parent set of T. Let $N(T)$ be the additive normal form of att$(C)T$, the product of the attributes inherited by C with T. In the event that T has an empty parent set, $N(T)$ is simply the additive normal form of T.

We claim that the following expression is left invariant by all of the rewrite rules:

$$\sum_{\textit{Non-virtual classes } C_i} att(C_i) \ + \sum_{\textit{Non-rewritten terms } T_j} N(T_j)$$

- *Rule 3.9 takes a term $T_j = 1$ with parent set $\{C_i\}$ and marks C_i non-virtual. The term att(C_i) is removed from the second sum, but added to the first sum.*
- *Rule 3.10 takes a term $T_j = A_1 \ldots A_n$ with parent set $\{C'\}$ and creates a new non-virtual class C inheriting $A_1, \ldots, A_n$. The term att$(C')A_1 \ldots A_n$ is removed from the second sum, but added to the first sum.*
- *Rule 3.11 takes a term $T = A_1 \ldots A_nT_1$ with parent set $\{C'\}$ and creates a new class C inheriting attributes $A_1, \ldots, A_n$ and att(C'). The term T_1 remains to be rewritten. The removal of T from the second sum, combined with the addition of T_1 to the second sum leaves the sum unchanged.*
- *Rule 3.12 takes a term $T = T_1 + T_2$ with parent set $\{C'\}$ and recursively performs two rewritings, for T_1 and T_2, each with parent set $\{C'\}$. The removal of T from the second sum, combined with the addition of T_1 and T_2 to the second sum leaves the sum unchanged, due to the distributivity of multiplication over addition (Lemma 2.2).*
- *Rule 3.13 is similar to the previous cases. One term is removed from the second sum, and replaced by three new terms. Distributivity again ensures that the net effect is that the sum is unchanged.*

Before any rewriting takes place, the result of the invariant is just the additive normal form of the initial expression. Once all terms have been rewritten, the result of the invariant is the sum of all attribute combinations at non-virtual nodes. Because the invariant is invariant, these expressions must be equal. The construction terminates because recursive rewriting steps are applied to subexpressions that are strictly smaller than the initial expression. □

Lemma 3.4. *The collection of Rewrite Rules 3.9, 3.10, 3.11, and 3.12 is complete, in the sense that every tuple-descriptor can be rewritten into a corresponding relational representation.*

Proof. *The rewrite rules recognize certain outermost operators, and decompose the problem into smaller subproblems. Each possible outer operator has a matching rewrite rule that applies. Tuple-descriptors are generated by a context free grammar in Definition 3.1. Let us show each production of the grammar is handled by the rewrite rules. We apply rewrite rules recursively, using an inductive argument on the number of operators in the expression. Given a tuple-descriptor T, rewrite rules are considered in the following order:*

- *If $T = 1$ apply Rewrite Rule 3.9.*
- *If T is a single attribute, or a product of attributes, apply Rewrite Rule 3.10.*

- *If T is of the form $A_1 \ldots A_n T_1$ apply Rewrite Rule 3.11.*
- *If T is of the form $T_1 + T_2$ apply rewrite rule 3.12.*

Additional Rewrite Rules, such as Rule 3.13 can be introduced to generate more efficient physical representations. However, Rules 3.9, 3.10, 3.11, and 3.12 are sufficient to handle any object-oriented tuple-descriptor. □

3.3 Faceted Physical Representation

The faceted data model represents objects in the domain of discourse as collections of clearly defined, mutually exclusive, and collectively exhaustive aspects, properties or characteristics [20].

Definition 3.2. *Let T be a tuple-descriptor. We say T is* purely faceted *if (a) it can be generated by the following context-free grammar:*

$$\begin{array}{lll} S & ::= & A \mid B \mid C \mid \ \ldots \ \mid SS \\ T & ::= & 1 \ \mid S+1 \mid TT \end{array}$$

and (b) in every subexpression of the form $T_1 T_2$ in T, the attributes appearing in T_1 are disjoint from the attributes appearing in T_2. □

A purely faceted tuple-descriptor contains no subtraction or addition (besides right-addition with 1). A *facet* is defined by a product of attributes, S in Definition 3.2. Each facet is optional and may appear in combination with any other facet. Because of the disjointness requirement, each facet F splits the universe of discourse into two mutually exclusive, collectively exhaustive sets of entities: those that have F and those that do not. While some tuple-descriptors, such as $A + B$, cannot be written in a purely faceted form, it is relatively straightforward to show the following lemma.

Lemma 3.5. *Every tuple-descriptor is equivalent to a tuple-descriptor of the form $T_1 - T_2$ where T_1 is purely faceted.*
Proof. *Given a tuple descriptor T, we may transform it into the form $T_1 - T_2$ using the following procedure. First, generate an additive normal form of T, call it T'. Next, generate a purely faceted T_1 by including a multiplicative term (A_i+1) for each distinct attribute A_i in T. Now expand T_1 into its additive normal form T_1'. Finally, generate $T_2 = T_1' - T'$, i.e., T_2 includes all additive terms that are there in T_1' but not in T'.* □

As an example of Lemma 3.5, $A + B$ would be expressed as $(A+1)(B+1) - (AB+1)$. Based on Lemma 3.5, one could rewrite expressions that are "close" to being faceted (i.e., the orthogonal composition of disjoint facets) as a purely faceted schema with the subtraction interpreted as a constraint on the allowed combinations of facets. Rewrite rules for an underlying faceted physical representation are analogous to Rewrite Rules 3.2 and 3.4 for relational schemas.

For an example of a faceted re-writing consider again Example 1.3, and its faceted factorization $(A + 1)(B + 1)(C + 1) - 1$. We first apply Rewrite Rule 3.4, that instructs us to construct a physical representation of $(A+1)(B+1)(C+1)$, and then add a constraint that disallows A, B, and C to all be null. We construct the physical representation of $(A + 1)(B + 1)(C + 1)$ as per Rewrite Rule 3.2, by creating a table with 3 columns, one for each attribute, and allowing each of A, B, and C to be null.

An alternative rewriting can use Rewrite Rules 3.5 and 3.8 to obtain three tables **IA**, **IB**, **IC**.

4. FACTORIZATION COMPLEXITY

Our formalism has some syntactic similarities to digital circuit design. However, some of the basic equivalence transformations of boolean circuit expressions are different from transformations in our algebra. For example, while in a digital circuit $X + Y + XY$ can be equivalently rewritten as $X + Y$, this equivalence does not hold for schema factorization. Nevertheless, some techniques demonstrated in this paper (e.g., Rewrite Rule 3.13), are similar in spirit to digital circuit optimization. Minimization of boolean expressions is NP-hard, with the exact complexity depending on the representation of the expression[19]. While schema factorization does not directly reduce to digital circuit optimization, we still expect the problem of finding the shortest expression for a schema to be difficult (likely exponential) in the general case.

Schema factorization also bears some similarities to algebraic polynomial factorization, with the difference that term constants and exponents are 0 or 1, due to idempotence of addition and of multiplication of attributes. Similarly to boolean and algebraic polynomial expressions, schema polynomials can be represented with a *kernel cube matrix (KCM)* [12]. Minimizing a polynomial expression represented by a KCM is equivalent to finding a maximum valued covering of the KCM, and is analogous to the minimum weighed rectangular covering problem described in [9], which is NP-hard.

5. FACTORIZATION POTENTIAL

In this section, we try to quantify the potential conciseness benefits of schema factorization. A complete design of a large real-world schema is beyond the scope of this paper. Nevertheless, we can assess the potential for reducing the schema complexity by taking an existing large schema, and measuring the decrease in schema redundancy obtained by applying some simple factorization heuristics. The schema we shall use is the data dictionary (i.e., catalog) of the Oracle 9i commercial database system [2]. We start with the additive normal form of the tables in the schema.

We use the number of attributes needed to write down the schema as our rough measure of schema complexity. This measure is both simple and independent of the physical representation of the resulting expression. The heuristics we employ do not change the number of addition operators, and do not increase the number of multiplications, so a reduction in the number of attributes represents a real reduction in the total size of the expression.

Heuristic 5.1. *Factor out all common attributes from two or more expressions in a sum, starting with expressions sharing the most common attributes.* □

Heuristic 5.2. *Factor out a single attribute from two or more expressions in a sum, starting with attributes appearing in the greatest number of expressions.* □

Heuristic 5.3. *Combine Heuristics 5.1 and 5.2: at each step, estimate which heuristic will produce a more compact expression at the next step, and choose that heuristic.* □

In Heuristic 5.3 we break ties in favor of Heuristic 5.2; this choice produced more compact results overall.

		Columns		Column Redundancy	
Schema	Tables	Total	Distinct	Average	Max
TAB	17	301	114	2.64	15
MVIEW	21	176	127	1.39	11
LOGMNR	37	313	140	2.24	29
REPCAT	43	324	187	1.73	20
ALL	446	3601	1621	2.22	148

Figure 2: Schema statistics

	Average Redundancy			Max Redundancy		
Schema	Heur. 1	Heur. 2	Heur. 3	Heur. 1	Heur. 2	Heur. 3
TAB	1.82	*1.51*	1.82	8	4	5
MVIEW	*1.17*	1.21	1.20	6	3	3
LOGMNR	1.59	*1.35*	1.39	12	6	6
REPCAT	1.43	*1.32*	1.32	10	5	5
ALL	1.63	*1.50*	1.54	59	9	24

Figure 3: Effect of heuristics on column redundancy

Example 5.1. *Consider again the expression*

$$ABCDE + ABCE + ABDE + ABE + AE + BE + E$$

of Example 3.1. This expression contains 5 unique attribute names, but it takes 21 occurrences of these attributes to write down the expression. Heuristic 5.1 transforms this expression into $ABCE(D+1) + ABE(D+1) + E(A+B+1)$, *bringing complexity measure down to 12. Heuristic 5.2 generates* $E(A(B(C(D+1)+D+1)+1)+B+1)$. *It now takes 7 attributes to write down the schema expression, a significant improvement. Heuristic 5.3 happens to produce the same expression as Heuristic 5.2 for this example.* □

We applied our heuristics to the schema as a whole (we refer to this as schema *ALL*), as well as to several sub-schemas, to highlight that different sections of the schema exhibit different degrees of redundancy, and can benefit from the use of our heuristics to different extent. We chose 4 sub-schemas of the Oracle data dictionary schema for our experiments: *TAB* (table management); *MVIEW* (materialized view management); *LOGMNR* (operation of the Oracle LogMiner utility); and *REPCAT* (replication).

We identified members of the sub-schemas based solely on table names. For example, tables that contain *TAB* in their name were included into the *TAB* sub-schema. Properties of the schema and its sub-schemas are summarized in Figure 2. *Average redundancy* is the total number of columns, divided by the number of distinct columns. For a non-redundant schema, this measure is 1. *Max redundancy* lists the number of times the most frequently used column appeared in the schema expression.

We summarize the result of applying heuristics to our schemas in Figure 3. Using any of the simple heuristics leads to a significant reduction in the average column redundancy. More sophisticated heuristics, such as those developed in [9, 12], may be used in addition to the heuristics described in this section.

Our heuristics act on existing schema expressions. Such processing is *retroactive* and cannot be expected to yield schemas as compact as those designed compactly from the start. However, the fact that we achieve significant improvement even in this manner highlights the potential of schema factorization to produce compact schemas.

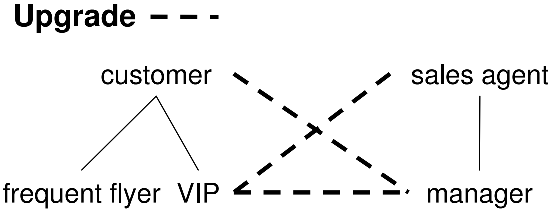

Figure 4: Ticket upgrade relationship

6. RELATIONSHIPS AND CONSTRAINTS

We now describe another application of schema polynomials – representing relationships with constraints. Relationship factorization utilizes a similar formalism as that described in the previous sections, but additionally accounts for an inheritance hierarchy over the entities. Relationship factorization can be applied as a second step over a factorized schema, or it can be used independently over an existing hierarchy.

Consider for example a schema that models the operations of an airline, and stores information about customers, sales agents, and upgrades. Figure 4 graphically represents this schema; class inheritance is denoted by solid lines, and relationships by dashed lines. In our example, certain customers have VIP status, which gives them preference for seat upgrades. Customers may participate in a frequent flyer program, making them eligible for air fare discounts. Customers purchase airplane tickets from sales agents, some of whom are managers. Any agent, manager or not, may offer an upgrade to a VIP customer free of charge. Managers may also use their discretion to upgrade non-VIP customers.

We propose to use schema polynomials to represent entities and relationships for such schemas. We use similar notation as in schema factorization, but shift gears in terms of semantics, and take a *class*, rather than an attribute, as the basic building block for modeling relationships. We use the symbols A, B, C, etc. to denote classes in the remainder of this section. These symbols are simply class labels, and do not describe internal class structure (e.g., class attributes or inheritance). However, we assume that the complete class hierarchy is provided and can be used to reason about the schema. The class hierarchy allows multiple inheritance and is acyclic.

6.1 Class Descriptors

Definition 6.1. *A* class-descriptor T is defined recursively using the following context free grammar:

$$T \quad ::= \quad 1 \mid A \mid B \mid C \mid \ \ldots \ \mid TT \mid T+T \mid T-T$$

The class-descriptor has set-based semantics. Concatenation is interpreted as set intersection[4], while addition and subtraction correspond to set union and difference, respectively. An approach to modeling inheritance with set expression has been described in [11]. We assume that there exists a single root for the class hierarchy, and denote this root class (i.e. the set of all objects) as 1. The special symbol 0 denotes the empty class. Addition, subtraction, and multiplication have the same properties as for tuple-descriptors in Section 2: addition and multiplication are commutative and associative; addition and subtraction are idempotent. 1 is a multiplicative identity, 0 is an additive identity and a right-identity for subtraction.

We assume that class membership respects the class hierarchy. Given classes T and S such that T is a subclass of S, an instance t of T is also an instance of S.

Definition 6.2. *An instance t matches a class-descriptor T if and only if:*

1. *$T = 1$.*
2. *T is a single class and t is an instance of that class.*
3. *$T = T_1T_2$ and t is an instance of both T_1 and T_2.*
4. *$T = T_1 + T_2$, and t is an instance of either T_1 or T_2.*
5. *$T = T_1 - T_2$, t is an instance of T_1 but not of T_2.*

In the airline example, we may denote the class of customers as C and the VIP customers sub-class as V; then instances of C who are not VIP are denoted as $C - V$. If L denotes frequent flyers, then the expression VL stands for the set of customers who are both VIP and frequent flyers, while $V + L$ denotes customers who are either VIP or frequent flyers.

We write $T_1 = T_2$ if the class-descriptors T_1 and T_2 always admit exactly the same instances, even if T_1 and T_2 are not syntactically identical. In other words, T_1 and T_2 are equivalent set expressions.

Definition 6.3. *A class-descriptor T_1* subsumes *another class-descriptor T_2 if every instance that matches T_2 also matches T_1. We denote this by $T_2 \subseteq T_1$.*

To determine whether a class-descriptor is subsumed by another, one can use standard subset inclusion reasoning from set algebra. Note that in presence of a class hierarchy $T \subseteq S$ if T is a sub-class of S. We assume that the class hierarchy is acyclic. Complexity of subsumption among arbitrary set expressions has been considered in the Set Constraints literature, and has been shown to be NP-complete if constants are allowed by the language [4].

Definition 6.4. *A class-descriptor is in* union-normal form *if it consists of a sum of subtractions of class-descriptors, where all multiplications are of basic class terms.*

Example 6.1. *Consider the following class descriptor:*

$$T = A(BC + (B - D)) + (ABC - D) \qquad (7)$$

T is not in union-normal form: $A(BC + (B - D))$ contains a nested multiplication and is equivalently re-written as $ABC+(AB-D)$. The full expression for T in union-normal form is then:

$$T = ABC + (AB - D) + (ABC - D) \qquad (8)$$

Definition 6.5. *A class-descriptor T is in* minimal union-normal form *if it is in union-normal form and if no pair-wise subsumption holds among class-descriptors in T.*

Lemma 6.1. *Every class-descriptor T has an equivalent class-descriptor in minimal union-normal form.*
Proof. *Transform the original class-descriptor into union-normal form by pushing nested multiplications outside, using the following transformations[5]:*

1. *$A(B + C) = AB + AC$*
2. *$A(B - C) = AB - C$*

Remove subsumed additive terms using subset inclusion reasoning in conjunction with the class hierarchy. The resulting descriptor is in minimal union-normal form. □

The union-normal form of expression in Example 6.1 is redundant, because $(ABC - D) \subseteq (AB - D)$. The minimal union-normal form of T is then:

$$T = ABC + (AB - D) \qquad (9)$$

Definition 6.6. *Given class-descriptors T_1 and T_2, the* least common subsumer *is the class-descriptor $lcs(T_1, T_2) = T_1 + T_2$.*

Transforming the lcs to minimal union-normal form may yield a more compact representation compared to the original expression. For example, $lcs(T_1, T_2) = T_2$ if $T_1 \subseteq T_2$.

We now explore how class-descriptors may be used to concisely represent relationships with constraints.

6.2 Relationship Descriptors

Definition 6.7. *An n-ary relationship is described by a* relationship-descriptor *R, generated using the following grammar (T refers to a class-descriptor in Definition 6.1):*

$$R \quad ::= \quad (T, \ldots, T) \mid R + R \mid R - R \mid RR$$

For an n-ary relationship the vector of class-descriptors $(T, \ldots, T)$ has exactly n elements. In all productions that combine relationship vectors, the vectors must agree in arity and produce a vector of that same arity.

We refer to addition, subtraction, and multiplication in the context of relationship-descriptors as *vector addition*, *vector subtraction*, and *vector multiplication*, respectively, in order to distinguish these operations from their counterparts over class descriptors.

[4] Note that semantics of multiplication is different compared to previous sections.

[5] Recall that we are using set semantics to represent inheritance. The correctness of these transformations is easy to verify using Venn diagrams.

Definition 6.8. *A* relationship instance *is a vector of class instances* $(t_1, t_2, \ldots t_n)$*. A relationship instance r matches a relationship descriptor R if and only if:*

1. $R = (T_1, \ldots, T_n)$*,* $r = (t_1, \ldots, t_n)$*, and each class instance* t_i *matches the respective class descriptor* T_i*.*

2. $R = R_1 + R_2$ *and r matches* R_1 *or r matches* R_2*.*

3. $R = R_1R_2$ *and r matches* R_1 *and r matches* R_2*.*

4. $R = R_1 - R_2$ *and r matches* R_1 *but not* R_2*.*

We write $R_1 = R_2$ if the relationship-descriptors R_1 and R_2 admit exactly the same relationship instances, even if R_1 and R_2 are not syntactically identical.

Lemma 6.2. *A relationship instance matches the descriptor* $(T_1, \ldots, T_n)(S_1, \ldots, S_n)$ *if and only if it matches the descriptor* $(T_1S_1, \ldots, T_nS_n)$*.*
Proof: Let us denote by $\mathcal{S}(R)$ *the set of relationship instances r that match R. By Definition 6.8,* $\mathcal{S}[(T_1, \ldots, T_n)(S_1, \ldots, S_n)] = \mathcal{S}(T_1, \ldots, T_n) \cap \mathcal{S}(S_1, \ldots, S_n) = \mathcal{S}(T_1S_1, \ldots, T_nS_n)$*.* □

Definition 6.9. *A relationship-descriptor* R_1 subsumes *another relationship descriptor* R_2 *if every relationship instance that matches* R_2 *also matches* R_1*.*

Lemma 6.3. *A single-term relationship-descriptor of the form* $R_1 = (T_1, \ldots, T_n)$ *subsumes another single-term relationship-descriptor* $R_2 = (S_1, \ldots, S_n)$ *if and only if for each position i, the class descriptor* T_i *in* R_1 *subsumes the class-descriptor* S_i *in* R_2*.*
Proof. *Since* $\mathcal{S}(R_2) \subseteq \mathcal{S}(R_1)$*, then for each relationship instance r, if* $r \in \mathcal{S}(R_2)$ *then* $r \in \mathcal{S}(R_1)$*. Since each class instance* r_i *in r matches the corresponding class descriptor* S_i *in* R_2*, then* $\mathcal{S}(S_i) \subseteq \mathcal{S}(T_i)$*.* □

Definition 6.10. *We say that a relationship-descriptor R is in* union-normal form *if R contains no vector multiplications, and consists of a vector-sum of subtractions of relationship-descriptors.*

Consider again the *Upgrade* relationship in Figure 4. We may represent this relationship in several equivalent ways, depending on the point of view of the user.

$$\begin{aligned} Upgrade_1 &= (A, V) + (M, C) + (M, V) \\ Upgrade_2 &= (A, C) - (A - M, C - V) \end{aligned}$$

The first variant enumerates all valid combinations of arguments, and states that *Upgrade* is a relationship between a sales agent and a VIP customer, between a manager and a customer, or between a manager and a VIP customer. The second variant states that *Upgrade* is a relationship between an agent and a customer, with the exception of agents who are not managers and customers who are not VIP.

Definition 6.11. *A relationship-descriptor R is in* minimal union-normal form *if it is in union-normal form, and if no pairwise subsumption holds among relationship-descriptors in R.*

In our running example, the expression $Upgrade_1$ is transformed into minimal union-normal form by removing the term (M, V) because it is subsumed by both other terms in the expression, by Lemma 6.3.

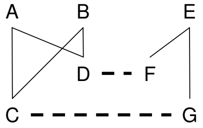

Figure 5: Example of a relationship

Relationship-descriptors might be thought of as data types for external applications. External applications typically accept simpler (usually atomic) data types for the attributes; we thus need a principled mechanism for transforming complex relationship-descriptors into such simplified form. We propose to do this by means of a *relationship signature*: a generalization of the least common subsumer over class-descriptors (see Definition 6.6) to relationship-descriptors. Relationship signature of an n-ary relationship R is obtained by applying $lcs(T_1, \ldots, T_n)$ to the positive terms in the descriptor for R. For example,

$$sig(Upgrade_1) = (lcs(A, M), lcs(V, C)) = (A, C) \qquad (10)$$

$Upgrade_2$ contains a single additive term, (A, C), which is trivially also $sig(Upgrade_2)$.

A signature provides a simpler, more concise view of a relationship; it summarizes the relationship by simplifying the types of its arguments. In the *Upgrade* example, the signature (A, C) simply states that *Upgrade* is a relationship between a sales agent and a customer. Note, however, that a relationship signature is an approximation, because it admits invalid combinations of agent and customer.

The signature of the *Upgrade* relationship produces atomic types for each argument. However, this may not always be the case. Consider the binary relationship in Figure 5: $R : (C, G) + (D, F)$. The signature of this relationship is $sig(R) = (C + D, F + G)$: both arguments are of union types. If the external application does not recognize union types, this signature requires further simplification: we need to find, for each argument, the least common subsumer that is atomic. Note that such signatures are not always unique: (A, E) and (B, E) are both valid atomic signatures for R.

7. RELATED WORK

In their seminal paper, Tsalos et al. [16] focus on physical data independence in the relational setting. Deriving alternative physical representations of XML schemas [1] based on cost considerations has been studied in the LegoDB project [3]. The aim of our work is to optimize the conceptual level representation, and then suggest rewrite rules that can translate the resulting schema into a physical representation. Representing schemas conceptually using polynomials allows the user to combine multiple design formalisms, leading to simpler, more compact schemas.

UML [13, 8] supports a rich set of design concepts, and gives multiple alternative design options. While the Schema Polynomials formalism does not support all features of UML, our work is the first attempt at providing conciseness-based design guidelines. In our framework, a design formalism will match a schema, or a part of a schema, if it can express the schema concisely.

The problem of minimizing concept descriptions has been considered in the Description Logics literature. Baader et al [6, 7] consider the problem of using a terminology τ to rewrite a concept definition C into an equivalent definition C' that is more concise than C. The motivation for such a rewriting is, like in our work, increased human readability. However, the rewriting methodology is different: terminology-based rewritings minimize concept descriptions one at a time using substitutions, while schema factorization allows for concise representations of schemas as a whole. In [7] the authors show that the size of a least common subsumer expression may be exponential in the size of a terminology.

Yu and Jagadish [21] developed a schema summarization framework, with the goal of concisely representing complex real-life schemas. In addition to generating concise representations, schema factorization can be used to generate multiple alternative representations, thus orienting the schema to the point of view of the user.

The idea of constructing a purely faceted representation and disallowing certain combinations of facets was used by Tzitzikas et al. [17, 18] to address a different kind of problem: finding compact representations of ontological term systems with subsumption relationships between terms.

Dagan and Itai [11] describe a set-based formalism for representing inheritance. We take a similar approach when reasoning about class and relationship-descriptors.

Buneman and Pierce [10] develop a type system for semistructured data that incorporates union types and propose a syntax that allows to operate on the type descriptors in their original compact form, avoiding a possible exponential explosion if the disjunction were unfolded.

8. CONCLUSIONS AND FUTURE WORK

We presented the schema factorization framework: a formal conceptual schema representation language that combines different design formalisms, and allows schema manipulation that exposes the strength of each of these formalisms. Schema factorization is a unifying model that can be used to transform schemas into simpler, more concise representations. The central benefit of schema factorization is in further decoupling physical and conceptual data representations: schema design is elevated to a conceptual level where it is no longer restricted by the idioms available to a particular design formalism.

We presented rewrite rules that can be used to translate conceptual schemas into their relational, object-oriented, and faceted physical representations. We demonstrated the potential of our approach by applying simple heuristics to a large real-life relational schema, and observing a significant improvement in schema compactness.

A schema polynomial can easily represent a logical schema with parts that are relational, object-oriented, or faceted in nature. Our physical representations have so-far focused on target platforms that are either purely relational or purely object-oriented. We plan to study combined platforms in future work. Object-relational databases provide a potential target platform. However, current object-relational systems have limitations, such as the choice to make objects subsidiary to tables, and the absence of multiple inheritance.

We introduced an application on schema polynomials to the representation of relationships with constraints. This approach allows for concise representation of relationships, and enables the user to orient the representation to his point of view. We plan to consider relationship composition in our future work.

In the future, we plan to consider how to extend schema factorization to include more constructs, such as overriding in the object-oriented model. We also plan to design efficient factorization algorithms, and to experimentally evaluate the potential of our methodology for schema integration.

9. REFERENCES

[1] Extensible markup language (XML). http://www.w3.org/XML/.

[2] Oracle9i database release 2 (9.2) documentation. http://www.oracle.com/technology/documentation/oracle9i.html.

[3] The unified modeling language user guide. 2005.

[4] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In *CSL*, pages 1–17, 1993.

[5] P. Atzeni and N. M. Morfuni. Functional dependencies in relations with null values. *Inf. Process. Lett.*, 18(4):233–238, 1984.

[6] F. Baader, R. K"usters, and R. Molitor. Rewriting concepts using terminologies. In *KR*, pages 297–308, 2000.

[7] F. Baader and A.-Y. Turhan. On the problem of computing small representations of least common subsumers. In *KR*, pages 99–113, 2002.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide.* Addison-Wesley, 2nd edition, 2005.

[9] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level logic optimization and the rectangular covering problem. In *ICCAD*, pages 66–69, 1987.

[10] P. Buneman and B. Pierce. Union types for semistructured data. In *DBPL*, pages 184–207, 1999.

[11] I. Dagan and A. Itai. A set expression based inheritance system. *Annals of Mathematics and Artificial Intelligence*, (4):269–280, 1991.

[12] A. Hosangadi, F. Fallah, and R. Kastner. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):2012–2022, 2006.

[13] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley Longman, 1998.

[14] M. Levene and G. Loizou. Axiomatisation of functional dependencies in incomplete relations. *Theor. Comput. Sci.*, 206(1-2):283–300, 1998.

[15] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 3rd edition, 2003.

[16] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB J.*, 5(2):101–118, 1996.

[17] Y. Tzitzikas, A. Analyti, and N. Spyratos. Compound term composition algebra: The semantics. *J. Data Semantics*, 2:58–84, 2005.

[18] Y. Tzitzikas, A. Analyti, N. Spyratos, and P. Constantopoulos. An algebraic approach for specifying compound terms in faceted taxonomies. In *EJC*, pages 67–87, 2003.

[19] C. Umans. The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, 63(4):597–611, 2001.

[20] B. Wynar. *Introduction to Cataloging and Classification*. Libraries Unlimited, Inc., 8th edition, 1992.

[21] C. Yu and H. V. Jagadish. Schema summarization. In *VLDB*, pages 319–330, 2006.